

# What Goes On Inside Windows 2000: Solving the Mysteries of the Loader

Russ Osterlund

Note: Several of the figures mentioned are located at the end of this article.

## SUMMARY

DLLs are a cornerstone of the Windows operating system. Every day they quietly perform their magic, while programmers take them for granted. But for anyone who's ever stopped to think about how the DLLs on their system are loaded by the operating system, the whole process can seem like a great mystery. This article explores DLL loading and exposes what really goes on inside the Windows 2000 loader. Knowing how DLLs are loaded and where, and how the loader keeps track of them really comes in handy when debugging your applications. Here that process is explained in detail.

Ever since I first encountered a definition of dynamic link libraries in a description of the then-new operating system OS/2, the idea of DLLs has always fascinated me. This beautifully simple concept of modules that could be loaded and unloaded as needed with well-defined interfaces that outlined routines written beforehand and, perhaps by other programmers, was a powerful jolt to me because I was more accustomed to statically linked code in mainframe or MS-DOS® programs. And, like many others new to programming for Windows®, the first utility I built enumerated DLLs that were already loaded into the system in order to demonstrate this concept at work. Now, even with the Windows world changing at a frenetic pace, employing COM interfaces and their ActiveX® components, and moving toward common language runtimes with their assemblies of managed code, the humble DLL remains at the center of things, providing services to the system on an as-needed basis.

During this long association with DLLs, I accepted their loading by the operating system as if it were magic and never truly appreciated the amount of work required by LoadLibrary and its variations. This article is an attempt to rectify that oversight by looking inside NTDLL.DLL. Since I do not have access to the source files, much of what I discuss here falls under that nebulous category of undocumented information and is therefore subject to change or obsolescence in future releases of the operating system.

The details that I'll cover are based on an examination of the binaries available when I wrote this article, Windows 2000 Professional (Build 2195: Service Pack 1). Access to a properly installed set of debug symbol files, .DBG and .PDB dated July 9, 2000, and working with a suitable debugger will make the information easier to understand.

This article can be seen as a precursor to Matt Pietrek's Under The Hood column in the September 1999 issue of MSJ, where Matt writes pseudocode for the operation of LdrpRunInitializeRoutines for Windows NT® 4.0 SP3 and describes how a library is initialized and when DiIMain gets called. Note that I will refer to this column frequently.

My discussion will begin with a brief look at LoadLibrary, starting with LdrLoadDll, and will conclude when LdrpRunInitializeRoutines is invoked. While trying to follow the execution path needed to load a simple DLL using a debugger, you can easily become confused by the numerous unconditional jump statements and lost in the recursion common in the later stages of DLL loading, so I'll guide you carefully through the call to LoadLibrary.

Note that all code modules mentioned in this article can be found at the link at the top of this article.

# What Goes On Inside Windows 2000: Solving the Mysteries of the Loader

Russ Osterlund

## All Paths Lead to LoadLibraryExW

There are several ways to get to LoadLibraryExW. For example, LoadTypeLibEx and CoLoadLibrary in the COM universe eventually call LoadLibraryExW. The two most familiar routes to LoadLibraryExW are LoadLibraryA and LoadLibraryW. All you need to do is specify one parameter—the name of the DLL—and you are on your way.

But if you examine a disassembly of LoadLibraryA and LoadLibraryW, you will discover that they are merely thin wrappers around the more versatile LoadLibraryExA and LoadLibraryExW APIs, respectively. With LoadLibraryA, there is a curious test for the DLL twain\_32.dll, but normally two zeroes are passed as the second and third parameters to LoadLibraryExA before continuing. LoadLibraryW is even more direct; the two zeroes are pushed onto the stack and the code moves directly onto LoadLibraryExW. These paths merge with an examination of LoadLibraryExA. There is a call to a helper routine to convert the DLL's name into a Unicode string before the code proceeds onto LoadLibraryExW.

LoadLibraryExW is a fairly involved routine which must decide between at least four different variations on the type of DLL-loading that the program wants to perform. In addition to the first parameter (which contains the name of the DLL), and the second parameter (which must be NULL according to the SDK documentation), there is a flag parameter that specifies the action to take when loading the module. You can ignore the flag value LOAD\_LIBRARY\_AS\_DATAFILE, since it does not lead to the desired goal: LdrLoadDll, an API exported by NTDLL.DLL. The remaining valid values—0, DONT\_RESOLVE\_DLL\_REFERENCES, and LOAD\_WITH\_ALTERED\_SEARCH\_PATH—all eventually find their way to LdrLoadDll.

It is interesting to note that there is no sanity check for the flag parameter that returns something like STATUS\_INVALID\_PARAMETER if values other than the documented ones are passed to LoadLibraryExW. For example, plugging in a reasonable value such as 4 results in a normal DLL load. In any case, the alternate paths taken with DONT\_RESOLVE\_DLL\_REFERENCES or LOAD\_WITH\_ALTERED\_SEARCH\_PATH will be noted later in this article. For now, I will concentrate on the normal case in which dwFlags has a value of 0.

There's one more API exported from NTDLL.DLL that leads to LdrLoadDll, but I have found neither documentation for it nor any references to it in existing DLLs. The name of the API is LoadOle32Export. It accepts two parameters, the image base for Ole32.DLL and the name of the routine to find, and it returns the address to the requested function. Whether this is a mysterious secret path or some kind of holdover from past versions of the operating system is uncertain.

## APIs Exported by NTDLL.DLL

Figure 1 lists the exported APIs in NTDLL.DLL beginning with the prefix "Ldr", and Figure 2 lists the internal routines beginning with "Ldrp". The main difference between the names for the LdrXXX and LdrpXXX routines is that the "p" indicates that these functions are private—hidden from the outside world. As you can see by examining the lists, familiar APIs such as GetProcAddress are wrappers around NTDLL exports like LdrGetProcedureAddress. In fact, many APIs in Kernel32 that are familiar to the SDK programmer, such as GetProcAddress, ExitProcess, ExpandEnvironmentStrings, and CreateSemaphore, pass the real work along to an NTDLL surrogate (LdrGetProcedureAddress, NtTerminateProcess, RtlExpandEnvironmentStrings\_U, and NtCreateSemaphore, respectively).

Now, let's take a close look at LdrLoadDll using DumpBin or my disassembler, PEBrowse (available from <http://www.smidgeonsoft.com>).

0X77F889A9 PUSH 0X1

# What Goes On Inside Windows 2000: Solving the Mysteries of the Loader

Russ Osterlund

```
0X77F889AB  PUSH      DWORD PTR [ESP+0X14]
0X77F889AF  PUSH      DWORD PTR [ESP+0X14]
0X77F889B3  PUSH      DWORD PTR [ESP+0X14]
0X77F889B7  PUSH      DWORD PTR [ESP+0X14]
0X77F889BB  CALL     0X77F887E0      ; SYM:LdrpLoadDll
0X77F889C0  RET      0X10
```

Do not be deceived by the apparent simplicity of this routine, which looks like merely a wrapper around the internal procedure, `LdrpLoadDll` (that I'll discuss in the next section). The first parameter points to a wide-string representation of the search path. The second parameter will hold a `DWORD` with the value of 2 if `DONT_RESOLVE_DLL_REFERENCES` was specified in the call to `LoadLibraryExW`. The third parameter contains a pointer to a Unicode string structure that encases the name of the DLL that is to be loaded. The fourth item is an output parameter and will receive the address at which the module was loaded when `LdrpLoadDll` has finished its work.

But what does the fifth parameter, hardcoded with a value of 1, mean? As you will see later, `LdrpLoadDll` can be called recursively in `LdrpSnapThunk`. Here, the value means normal processing, but later you will see that a value of 0 means process forwarded APIs.

## LdrpLoadDll

I have provided a small project in the code download for this article that contains a main executable, ingeniously named `Test`, and three DLLs: `TestDll`, `Forwarder`, and `Forwarded`. The DLLs demonstrate different variations that illustrate some of the scenarios `LdrpLoadDll` will commonly encounter. See the `Readme.txt` file in the code download for important information about setting environment variables and predefined breakpoints.

Figure 3 shows some of the internal loader routines you will bump into when you pass one of the documented flags to `LoadLibraryExW`. If you concentrate for a moment on the typical situation (#1 in Figure 3), you will see that there are six subroutines called directly by `LdrpLoadDll`: `LdrpCheckForLoadedDll`, `LdrpMapDll`, `LdrpWalkImportDescriptor`, `LdrpUpdateLoadCount`, `LdrpRunInitializeRoutines`, and `LdrpClearLoadInProgress`. (I'll discuss the first four subroutines later in this article.) `LdrpRunInitializeRoutines` has already been described in Matt Pietrek's column, so I won't go into it here. `LdrpClearLoadInProgress` is briefly mentioned in that column as well.

Let's take a high-level look at the steps taken by `LdrpLoadDll`, which occur as follows:

- Check to see if the module is already loaded.
- Map the module and supporting information into memory.
- Walk the module's import descriptor table (that is, find out what other modules this one is adding).
- Update the module's load count as well as any others brought in by this DLL.
- Initialize the module.
- Clear some sort of flag, indicating that the load has finished.

Looking more closely at the routines listed, you will notice that `LdrpCheckForLoadedDll` can be and is called from several locations. It is therefore reasonable to assume that this is a helper function. Also, note that `LdrpSnapThunk` and `LdrpUpdateLoadCount`, as well as the previously mentioned case for `LdrpLoadDll`, are all candidates for recursion.

The code for `LdrpLoadDll.cpp` contains pseudocode for the operations found in this and the other loader routines. My pseudocode is not a copy of the actual code for `LdrpLoadDll` and should not be compiled; it represents intelligent guesswork obtained by studying a disassembly of the routine. It also

# What Goes On Inside Windows 2000: Solving the Mysteries of the Loader

Russ Osterlund

shows how much information can be brought together about any section of code just by taking the time to study the available binaries. The pseudocode by no means covers every detail.

In many cases, the variable names were assigned based on their role in the code. Others were taken from parameter descriptions found in Windows NT/2000 Native API Reference by Gary Nebbett (New Riders Publishing, 2000). Take a look at this book to see what takes place inside NTDLL.DLL. It provides a reference for the Nt/Zw routines—the so-called native APIs—and where possible relates these routines back to their Win32® counterparts.

Now I'll describe in some detail what happens when your code loads the typical DLL, then I'll throw in a few options for variety. LdrpLoadDll first sets up a `__try/__except` block, then checks the flag, `LdrplnLdrlnit`. If the flag is turned on, then it sets up a critical section block to prevent updates in the data structures that it will be referencing and modifying. Next, it checks on the length of the incoming DLL's name against the maximum, 532 bytes or 266-wide characters, which is close to the better-known constant `MAX_PATH` and its value of 260. If the length of the name exceeds the maximum value, then the routine quits with a return code of `STATUS_NAME_TOO_LONG`.

At this point, LdrpLoadDll attempts to locate the position of the period character (dot) in the module's name. If the dot is missing, then the routine appends the string ".dll" to the end of the module's name, assuming this does not exceed the maximum size. There are two items to note here: first, you do not need to pass the .dll extension in your calls to LoadLibrary, and second, some of the features, such as the API-forwarding discussed later in this article, will not work with any extension other than .dll. (For a preview, try the following: change the output name of the Forwarded project to "Forwarded.cpl" and run the test program, making certain that the GetProcAddress stuff is included. You'll see that the GetProcAddress call fails and the messagebox that should display "Hello World!" will not appear!)

The test for ShowSnaps and enabling this handy debugging aide on your system were explained in Matt's column. ShowSnaps occupies the second bit position in the Registry value GlobalFlag, located at `HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Session Manager`. When enabled, this provides feedback in a debugger's output window about actions the loader is taking. If you decide not to enable "snaps" on your system, then the text strings appearing throughout the code will provide useful diagnostic hints. (In the download is an ASCII text file, `LdrSnap(Forwarder).TXT`, that includes the snap outputs produced by loading the Forwarder test DLL. Later in the article you will find figures showing this information for Forwarded.DLL and TestDll.DLL.) LdrpLoadDll then constructs a Unicode string of the module's name that will be used by the first subroutine, LdrpCheckForLoadedDll.

## LdrpCheckForLoadedDll

The pseudocode for this routine can be found in `LdrpCheckForLoadedDll.cpp`. You'll see that the code checks all of the possible lists for loaded DLLs and if the initial search fails, the routine tries the alternative. Following all of the paths in this routine is rather complicated, but the essence of this routine is rather simple to understand. There are two places to start a search: an optimization based on a hash table found inside of NTDLL.DLL, and walking the module list maintained inside the process's environment block (PEB). The definition of the module list and its location in memory will follow shortly. LdrpLoadDll, the calling routine, starts the search by setting the `UseLdrpHashTable` parameter to 0. Later on, I'll show you a similar case in which the parameter is set to 1 and the hash table is used. If the `DllName` parameter does not contain a path, the comparison using `RtlEqualUnicodeString` is made on the simple file name entries.

There is a small wrinkle in the search option when the hash table is not used, however. If the incoming `DllName` contains a path, LdrpCheckForLoadedDll calls on `RtlDosSearchPath_U` to validate the path, then examines the entries in the module list that contain fully qualified file names. In case

# What Goes On Inside Windows 2000: Solving the Mysteries of the Loader

Russ Osterlund

you were wondering how the Microsoft® .NET Common Language Runtime (CLR) supports side-by-side execution of different versions of the same assembly, this information should give you a head start. If the module is found in the PEB list but contains a load count of 0, `LdrpCheckForLoadedDll` forces the load by issuing a sequence of commands similar to what will be seen in `LdrpMapDll`. But describing this sequence now would be jumping too far ahead.

If the search has succeeded and the loading DLL is already part of the current process, then `LdrpLoadDll` has a few remaining details to take care of. The loaded count for this image will be incremented if the module is a DLL and if the loaded count field has not been set to -1 (which seems to mean "do not update the load count.") The module flag will also be cleared of its load-pending bit, which was set by the routine `LdrpUpdateLoadCount`. `LdrpLoadDll` will then leave the critical section block and set the `ImageBase` parameter to the address where the DLL was loaded.

More than likely, though, this is not the quick exit you are looking for—unless you are wasting machine cycles by using `LoadLibrary` to return an `HINSTANCE` of an already loaded module. Otherwise, `GetModuleHandle` is a better alternative. Instead, let's see what happens when the search fails.

## LdrpMapDll

Now that `LdrpLoadDll` knows that the requested DLL has not already been loaded into the process, it calls upon its second helper routine, `LdrpMapDll`, to perform the duties of finding the DLL's housing (the actual file), loading the DLL into memory (but not initializing it), and creating and adding a structure that I have tagged, `MODULEITEM`, to the PEB's module list. First, orient yourself in Figure 3 and become familiar with the support routines you can see in `LdrpMapDll`, (`LdrpCheckForKnownDll`, `LdrpResolveDllName`, `LdrpCreateDllSection`, `LdrpAllocateDataTableEntry`, `LdrpFetchAddressOfEntryPoint`, and `LdrpInsertMemoryTableEntry`). In the file `LdrpMapDll.cpp`, you will find the pseudocode that shows the fine points of what actually takes place here.

`LdrpCheckForKnownDll` checks to see if the loading DLL can be located in the directory specified in the Unicode string, `LdrpKnownDllPath`, located at `0x77FCE008`. Examining this memory location with a debugger reveals that this variable contains the value `"C:\WINNT\SYSTEM32"` (or something similar, depending upon how your system is configured).

If you start up `WinObj` (found at <http://www.sysinternals.com>) or my utility, `NtObjects` (found at <http://www.smidgeonsoft.com>), and select `View | Executive Objects`, you will find an object directory called `KnownDlls`. Under this directory, you will see the item `KnownDllPath` (a symbolic-link object) that contains the value for your system. `LdrpCheckForKnownDll` allocates two Unicode strings to hold the fully qualified DLL file name and the file name by itself. By calling `NtOpenSection` with the fully qualified file name, a `FileExists` operation is performed and `LdrpCheckForKnownDll` returns a section handle if the DLL is known. Otherwise, the two Unicode strings are freed and `LdrpMapDll` must call on the services of `LdrpResolveDllName` and `LdrpCreateDllSection`.

`LdrpResolveDllName` returns two Unicode strings, the loading DLL's file name and its fully qualified file name. If a search path was not provided, the routine uses the path found at the hardcoded address `0X77FCE30C` in `NTDLL.DLL`, which points to a default search path. `RtlDosSearchPath_U`, though, performs the real work in this routine. If `RtlDosSearchPath_U` can find the loading DLL, it returns the length of the path where it resolved the DLL's name.

If you try to load a DLL that cannot be found in the search path, the search returns 0. `LdrpMapDll` then responds to this result and returns with the status code `STATUS_DLL_NOT_FOUND`, which leads to

# What Goes On Inside Windows 2000: Solving the Mysteries of the Loader

Russ Osterlund

a quick exit from LdrpLoadDll. If you change the test program so that it tries to load a DLL with the name "bogus," you can check this for yourself.

Assuming all is well and the loading DLL has been found, LdrpCreateDllSection must take over and create the all important section handle. Since sections are classified as kernel objects, the path needs to be converted into something that the Executive Object Manager can understand. This is where the routine RtlDosPathNameToNtPathName comes into play. It takes a fully qualified file name "C:\Projects\LoadLibrary\Debug\TestDll.dll" and returns something like "\??\C:\Projects\LoadLibrary\Debug\TestDll.dll." If the file name cannot be interpreted, then the routine returns STATUS\_OBJECT\_PATH\_SYNTAX\_BAD and ends execution.

LdrpCreateDllSection is just a thin wrapper around the so-called native API, NtCreateSection. First, a file handle is obtained with NtOpenFile. If that call fails, then a return code of STATUS\_INVALID\_IMAGE\_FORMAT is generated. Otherwise, the Object Manager creates the section handle and the file handle is closed via NtClose.

Now that LdrpMapDll has the section handle, it can actually load the DLL into the process's address. The DLL is brought in as a memory-mapped file through the services of NtMapViewOfSection. First, the defaults are set for the base address and the size of the mapping object—with the latter, a value of 0 indicates to the system that the entire section object will be mapped. Then, a field in the PEB reserved for subsystem calls is loaded with the value of the fully qualified file name.

Those of you who have written debug loops and have handled the debug event LOAD\_DLL\_DEBUG\_EVENT may be interested to know that the lpImageName field of the LOAD\_DLL\_DEBUG\_INFO structure is filled with the contents of this field, which is reserved for subsystem calls in the process that's being debugged.

Now, NtMapViewOfSection is called. This triggers the notification of the loading DLLs as seen in debuggers like the one in Visual C++®. LdrpMapDll restores the previous value of the PEB's subsystem data field and checks on the success of the operation. If the DLL has been successfully mapped, LdrpMapDll now possesses an actual memory address with which it can work. How NtMapViewOfSection returns the image base when no hint was passed to it is a question that could be resolved with further exploration.

## PEB Load List

After a sanity check on the image now mapped into memory, LdrpMapDll continues with some bookkeeping. Earlier I mentioned a data structure I named MODULEITEM that is created and added to the process's PEB. In the code for LdrpLoadDll.h you will find a reconstruction of this structure. The job of LdrpAllocateDataTableEntry is to allocate this object and initialize the ImageBase field using three values: the HMODULE handle returned by NtMapViewOfSection, the ImageSize field using the SizeOfImage value found in the portable executable (PE) file's optional header, and the TimeDateStamp from the equivalent field in the PE file header. If the memory allocation fails, the routine returns a NULL pointer. LdrpMapDll then removes the file mapping, closes the section handle, and fails, returning STATUS\_NO\_MEMORY.

Assuming that all is well, the LoadCount field in the newly built ModuleItem is zeroed out and ModuleFlags gets initialized. (LdrpLoadDll.h provides the possible values for this field.) The two Unicode string fields containing the full path to the DLL and the file name are filled in. Then a call placed to the small helper routine, LdrpFetchAddressOfEntryPoint, inserts the structure's EntryPoint field.

# What Goes On Inside Windows 2000: Solving the Mysteries of the Loader

Russ Osterlund

With the `ModuleItem` partially initialized, `LdrpMapDll` calls `LdrpInsertMemoryTableEntry` to insert the entry into the process's module list located in the PEB. At offset `0x0C` in the PEB, which can usually be found at the address of `0x7FFDF000`, you will find the pointer to a structure I have named `MODULELISTHEADER`. Windows NT and Windows 2000 maintain three doubly linked lists that describe the load, memory, and initialization order for the modules in a process.

If you have ever used the PSAPI routines or the newly available `ToolHelp32` functions in Windows 2000 to enumerate the modules loaded in a process's address space, you should be having an epiphany. You have an alternate method to gather this information, in three different flavors, by using `ReadProcessMemory` and these undocumented structures. These structures have remained stable from Windows NT 4.0 through Windows 2000, and through the betas for Windows XP that were available when I did the research for this article. `LdrpInsertMemoryTableEntry` adds the `ModuleItem` structure to the end of the load and memory chains and fixes up the links appropriately. In my experience, I have found that the load and memory chains possess the same order; only the initialization list varies with its own order of the modules. See Matt Pietrek's column for more information on the initialization chain.

Returning to `LdrpMapDll`, you can now see the `ModuleFlags` field receiving some additional attention. If the executable is not `IMAGE_FILE_LARGE_ADDRESS_AWARE` and it is not of type `IMAGE_DLL`, the `EntryPoint` field gets zeroed out. `LdrpMapDll` tests the return value from `NtMapViewOfSection` to determine if the image was loaded at its preferred image base. Unfortunately, scenarios in which your DLL is parked in a different location is beyond the scope of this discussion, but now you know the way, so you can investigate this phenomenon on your own.

Finally, after a validation of the image on multiprocessor systems, `LdrpMapDll` closes the section handle obtained from `LdrpCreateDllSection` and returns with the results of its work. The validation only proceeds if your DLL contains data in the directory `IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG` and the `LockPrefixTable` field in this directory is nonzero. (NTDLL and Kernel32 contain the `IMAGE_DIRECTORY_ENTRY_LOAD_CONFIG` directory.) You can use `PEBrowse` to examine this directory, and those of you fortunate enough to have a multiprocessor machine can continue your own exploration down this path.

## LdrpWalkImportDescriptor

You may think that the route through `LdrpLoadDll` has been relatively straight and uncomplicated so far. But there still may be a maze of passages ahead. Your attempt to load `LdrpLoadDll` may have generated the need for additional modules, and this is where `LdrpWalkImportDescriptor` comes in. (In order to better understand my pseudocode, it will help to have some knowledge of the PE header definitions found in `WinNt.h`, especially those relevant to imports and exports. You can take a look at "Inside Windows: An In-Depth Look into the Win32 Portable Executable File Format" by Matt Pietrek in the February 2002 issue of MSDN Magazine.

Your typical module load takes you through the twists and turns of `LdrpWalkImportDescriptor`. However, if you specify `DONT_RESOLVE_DLL_REFERENCES` in your call to `LoadLibraryExW`, then you will avoid the upcoming maze. You should read the SDK documentation carefully to make certain that this is what you want. There is also a mechanism, which I'll explain later, to help you avoid the loops and recursion that are part of `LdrpSnapIAT` and `LdrpSnapThunk`. But if you don't take either of these alternatives, then you need to know what happens with `LdrpWalkImportDescriptor`.

Orient yourself once again by reexamining Figure 3, locating `LdrpWalkImportDescriptor`. `LdrpWalkImportDescriptor` has two subroutines: `LdrpLoadImportModule` and `LdrpSnapIAT`. This does

# What Goes On Inside Windows 2000: Solving the Mysteries of the Loader

Russ Osterlund

not seem so bad, but one tip-off that this code will soon become interesting is that there are four nesting levels in the routines for LdrpSnapIAT. The number and depth of nested functions is one metric that indicates the complexity of code. You should take note that recursion is possible in not one, but two locations in LdrpSnapIAT. You may recall that in the section on APIs exported by NTDLL.DLL I mentioned the apparent simplicity of the call to LdrpLoadDll and the fifth parameter that took a 0 or a 1. LdrpSnapIAT can also be recursive inside LdrpGetProcedureAddress. Finally, to make things even more complex than they already were, it's possible that a typical DLL may import other modules that start a cascade of additional library loads. The loader will need to loop through each module, checking to see if it needs to be loaded and then checking its dependencies.

With that in mind, let's take a look at the pseudocode found in LdrpWalkImportDescriptor.cpp. (If you are following along with the debugger, change the test program to load the Forwarded.DLL module and restart the debugger.) Execution starts with two calls to RtlImageDirectoryEntryToData to locate the Bound Imports Descriptor and the regular Import Descriptor tables. For the moment, ignore the call for that bound import thing except to notice that the code checks for its presence first. (I'll discuss binding later.) In Forwarded.DLL, LdrpWalkImportDescriptor detects two imported modules, User32.DLL and Kernel32.DLL, and now calls upon LdrpLoadImportModule for assistance.

LdrpLoadImportModule constructs a Unicode string for each DLL found in the import table and then employs LdrpCheckForLoadedDll, using the hash table in NTDLL that was mentioned earlier to see if they have already been loaded. Note that the call here is made with only the file name (no fully qualified path) and the process's search path. If you have ever had your application complain that it cannot find a DLL, you should realize that it may not be the loading module's fault. Check to see that all of its dependent modules can be found. If a module is found and already loaded, life is good because there is one less module to worry about. If it has not been loaded, then you've found an instance of recursion. A call to LdrpMapDll to bring the DLL into the process' address space is followed by a call to LdrpWalkImportDescriptor. Now you're in the middle of the twisting mazes I mentioned.

## The IAT and Forwarded API Processing

Once LdrpWalkImportDescriptor knows that the module is in memory (either through a call to LoadLibraryExW way back at the beginning or via the LdrpMapDll call in LdrpLoadImportModule), the next step is to examine each and every API referenced in Forwarded's imported module list with the call to LdrpSnapIAT. Why is this necessary? There are two reasons. First, you want to replace the placeholders in the Import Address Table (IAT) with real entry points. Second, you need to locate and process APIs that may have been forwarded on to another DLL. (Forwarding is one technique Microsoft employs to expose a common Win32 API set and to hide the low-level differences between the Windows NT and Windows 9x platforms.)

You may have observed either in the debug output strings or by carefully stepping through the loader code that at some point during Kernel32 processing, a check on NTDLL.DLL was made. Since just about any DLL with some executable code contains references to Kernel32, you may wonder why this is happening at all. You already know that NTDLL.DLL is loaded into every process. The answer is that Kernel32 contains APIs that are "forwarded" to NTDLL. Refer to Figure 4 for a complete list of these APIs for Kernel32 (version 5.0.2195.1600).

You may also notice that the list contains functions that are common requirements for just about any kind of serious programming. Remember, LdrpLoadDll needs to load every module referenced by the loading DLL, including those "hidden" references contained in forwarded APIs. Thus, a reasonable conclusion from all of this is that an import table walk will take place every time you load a DLL—at

# What Goes On Inside Windows 2000: Solving the Mysteries of the Loader

Russ Osterlund

least for Kernel32's forwarded APIs and for any additional forwarded APIs you may have decided to include in your application!

To allow you to experiment with this concept, I have included Forwarder.DLL and Forwarded.DLL in the download, as I've mentioned previously. The code for Forwarder.DLL is extremely simple—a DllMain with an export pragma. If you run DumpBin or PEBrowse on this module, you will see that it exports only one routine, and that the routine is marked with a designation that it is forwarded on to Forwarded.ShowMessage.

Now, turn your attention to Forwarded.DLL and take a look at what it exports. You will see a typical DLL with a DllMain and the single API, ShowMessage. However, if you changed the test program to load Forwarder and then observed what happened in the debugger, you might have wondered why no reference to Forwarded appeared. You will encounter this confusion unless you also included the GetProcAddress statement in your compile. Make certain that you include the GetProcAddress line and you will now see that both DLLs are loaded. You're observing another form of delay-load occurring and, perhaps, another example of the optimization that has been done on the loading engine in Windows.

Returning to LdrpWalkImportDescriptor, you'll find that it tests several items before calling LdrpSnapIAT. (For the purposes of following the next few steps, change the sample back to use Forwarded.DLL.) If you look up Section 6.4.1 of the Portable Executable Specification (found on the MSDN Library CD under Specifications), you will read that the time/date stamp field of an import directory will contain a value of 0 unless the DLL has been bound. One of the fields tested is the time/date stamp field, which is 0 in Forwarded.DLL, so let's step into LdrpSnapIAT.

LdrpSnapIAT wraps its execution around a `__try/__except` block first before locating the IAT in Forwarded.DLL, and then hunts for the export directory in the module that Forwarded is attempting to load (assume it is Kernel32 for now). It then changes the memory protection on the IAT of Forwarded.DLL to `PAGE_READWRITE` and proceeds to examine each entry in the IAT. (If you are able to examine the protection for this chunk of memory, you will see that it is normally `PAGE_READONLY` for your executables.) Going a bit further, you'll encounter LdrpSnapThunk.

LdrpSnapThunk requires an ordinal to locate an entry point and to determine whether or not the API is forwarded. If the hint value in Forwarded.DLL's import directory is correct, you can use that (generally, I have found this not to be the correct value). Otherwise, LdrpSnapThunk calls on the services of the helper routine, LdrpNameToOrdinal, to look up the correct value. Observe that LdrpNameToOrdinal uses a binary search on the export table to quickly locate the ordinal—more optimization in the loader—and note that the table must be sorted in alphabetical order for the search to work.

Now that you have an ordinal, you can look up the entry point for the API in Kernel32. LdrpSnapThunk first plugs the loading module's IAT entry with an address derived from the export table for Kernel32; see Section 6.4.4 of the PE specifications (which can be found on the October 2001 MSDN CD under Specifications | Microsoft Portable Executable and Common Object File Format) for more information. (This explains why the page protection was changed in LdrpSnapIAT.)

Section 6.3.2 of the PE specifications says:

**If the address (the entry-point) is not within the export section (as defined by the address and length indicated in the Optional Header), the field is an Export RVA: an actual address in code or data. Otherwise, the field is a Forwarder RVA, which names a symbol in another DLL.**

# What Goes On Inside Windows 2000: Solving the Mysteries of the Loader

Russ Osterlund

So now you can finally decide whether or not the API has been forwarded. In the vast majority of cases, the API is not forwarded, but let's assume you are looking at Forwarded's reference to HeapAlloc. (Check the math first. Kernel32's image base (0x77e80000) + HeapAlloc's entry point (0x0005b658) is 0x77edb658, which is inside the range for the export table, 0x77ed5c20 to 0x77edb770.)

LdrpSnapThunk now proceeds to break apart the forwarded reference for HeapAlloc, which will have the format NTDLL.RtlAllocateHeap, and then calls LdrpLoadDll to obtain NTDLL's image base—hmm, this looks like you're back at the beginning. But note that the fifth parameter is passed with a value of 0. Also note that the DLL name that was constructed before making the call to LdrpLoadDll lacks the .DLL extension. Fortunately, the call to LdrpLoadDll will succeed when LdrpCheckForLoadedDll figures out that NTDLL.DLL is already loaded.

But do you remember that experiment where I changed the extension for Forwarded to .cpl? Try this again and you will see that LdrpLoadDll now fails on the LdrpMapDll call with a STATUS\_DLL\_NOT\_FOUND return code. Now I have an explanation for the earlier results. With the module name out of the way, LdrpSnapThunk grabs the API name, RtlAllocateHeap, and forges on to LdrpGetProcedureAddress.

At this point, I am getting close to the end of this forwarded API processing, I promise. LdrpGetProcedureAddress is another routine that wraps its processing around a \_\_try/\_\_except block. The routine determines what type of API information it has been handed, either an ordinal if the API name parameter is null, or the name itself. The test is needed in order to properly set up parameters for an upcoming call to LdrpSnapThunk. But wait a moment. Didn't LdrpSnapThunk just bring us to this point? The two parameters are a pointer to the API's entry in the IAT and an overloaded item that contains either the image base of the loading DLL or a pointer to an IAT entry. If the flag, LdrpInLdrInit, is turned on, the process's critical section is entered. And now let's really dive in deep and step into LdrpCheckForLoadedDllHandle.

Fortunately, the functionality here is pretty simple to describe and understand. I need a MODULEITEM before I can continue. LdrpCheckForLoadedDllHandle first examines a handle cache residing at LdrpLoadedDllHandleCache to see if the image base there is the same as its input parameter, hDll. If not, the routine perseveres by walking the LoadOrder list, searching for the MODULEITEM whose image base matches hDll and whose linkage in the memory order list has been established. Once it finds an entry matching these criteria, it updates the cache with it and hands back to LdrpGetProcedureAddress the MODULEITEM that it found, or returns a 0 to indicate failure.

With a MODULEITEM in hand, LdrpGetProcedureAddress now calls an old friend, RtlImageDirectoryEntryToData, to locate the item's export directory and starts that twisty, recursive call to LdrpSnapThunk. What is going on here? Why is this call necessary? The answer, in part, is that this new API itself may be forwarded! I am not aware of such a situation in Windows 2000, but the possibility certainly exists. Happily, HeapAlloc's processing ends with RtlAllocateHeap inside of NTDLL, and LdrpSnapThunk returns an IAT entry with the entry point to this API. LdrpGetProcedureAddress frees up any work areas it might have created, exits the critical section (if it was acquired), and returns. Whew!

Next, LdrpSnapThunk checks the return code and returns STATUS\_ENTRYPOINT\_NOT\_FOUND if the API was not found. Otherwise, it replaces the entry in the IAT with the API's entry point and continues on. Study Section 6.4.4 in the PE specifications and especially the references to binding for a more complete picture of what is happening.

# What Goes On Inside Windows 2000: Solving the Mysteries of the Loader

Russ Osterlund

Now let's return to LdrpSnapIAT and move on to the next imported API in Kernel32 (or break from the loop if the LdrpSnapThunk call failed). Once all of the entries are processed in Kernel32's import table, LdrpSnapIAT restores the memory protection it changed at the beginning of its work, calls NtFlushInstructionCache to force a cache refresh on the memory block containing the IAT, and returns back to LdrpWalkImportDescriptor. The cache refresh might be a little surprising, but in many executables the IAT can be found in the .text section where code is found. If LdrpWalkImportDescriptor does not flush the memory block containing the updated IAT, then all of the previous work will have been for naught because the processor may continue to use the old version of the memory block. (For more information read the SDK documentation for the Kernel32 API FlushInstructionCache, which is just a thin wrapper around NtFlushInstructionCache.)

If you want to see the results of RunTime Binding via LdrpSnapIAT and LdrpSnapThunk on the IAT for Forwarded.DLL (SP1), take a look at Figure 5.

## Bound DLL Processing

You may recall that LdrpWalkImportDescriptor tested for the existence of two directories or descriptors, the regular Import Descriptor and something called the Bound Imports Descriptor, and tried to use the Bound Imports Descriptor first if it was available. Also, LdrpSnapIAT examined the time/date stamp in the Import Directory Table for a special value of -1 before moving on to LdrpSnapThunk. There just may be an alternative to all of this import table munging by pre-binding your DLL. Now is the time to change my test project to load TestDll and see what happens in the Windows 2000 loader with a module that has been bound ahead of time.

If you have not created the environment variable "MSSdk" as I mentioned in the readme.txt file, and set it equal to the root directory where your Platform SDK is located, do so now and rebuild TestDll (a post-link step should kick in that performs the binding operation). Or, from a command line you can enter and run the following command:

```
bind -u testdll.dll
```

When you examine the resulting executable, you should see that a new directory in the optional header array has been filled: the slot corresponding to IMAGE\_DIRECTORY\_ENTRY\_BOUND\_IMPORT. Launching the result in the debugger, you will also observe that the tests for the Bound Imports Descriptor will succeed. Try the test without a call to GetProcAddress on "fnTestDll" and you will see that when LdrpWalkImportDescriptor issues its call to LdrpLoadImportModule, the check for an already loaded module (Kernel32.DLL) will succeed and that means you can avoid the nasty bumps and turns that made the code very complex earlier in the discussion of LdrpWalkImportDescriptor. My DLL loads faster because there is no looping through the APIs I imported from Kernel32 since the fix-ups have already been done (including the forwarded references). I feel like I've found the Holy Grail.

But this old cup loses some of its shine when I change the sample to call fnTestDll. Before continuing, see if you can foretell why you will be walking that twisty maze again. The reason is that Forwarded.DLL, the module that contains the real code for my forwarded API, fnTestDll, was not itself bound. Run the bind utility on Forwarded.DLL and the brilliance of my newfound treasure returns. The moral of this little exercise is that in order to gain the full measure of efficiency that pre-binding a module provides, make certain that all the subordinate modules have been bound, too.

There is a slight downside to pre-binding a DLL, though. What happens when the next version of the operating system appears with a new version of Kernel32 and new locations for the exported

# What Goes On Inside Windows 2000: Solving the Mysteries of the Loader

Russ Osterlund

functions? Or consider the consequences of another module loading and occupying the slot reserved in memory for that DLL you have bound your executable to. Your bindings, the hardcoded addresses in the IAT, will be incorrect and considered stale by the loader. Under these conditions, the binding is effectively ignored, the LdrpWalkImportDescriptor processing takes place, and you are no better off than you were before.

On the other hand, if you can manage to keep your DLLs in sync with the current versions of the system DLLs and any others you may use, you should see an improvement in your module loads. As the SDK documentation states: "You can minimize load time by using Bind to bypass this lookup." (For more discussion concerning BIND.EXE and other load issues, see Under the Hood in the May 2000 issue of MSDN Magazine. Also, note that the system DLLs, Kernel32, GDI32, User32, AdvApi32, and so on have been pre-bound.) Figure 6 shows the results of pre-binding using the SDK Bind utility on the IAT for TestDll.DLL (SP1).

## LdrpUpdateLoadCount

If you take stock of what you have seen and learned so far, you will realize that the first three parts in LdrpLoadDll's processing have been completed. The last part of LdrpLoadDll to explore involves an update to module reference counts. That is the job for LdrpUpdateLoadCount.

LdrpUpdateLoadCount is a dual-purpose routine; it is called when the DLL is both loading and unloading. It attempts to walk either the Bound Imports table or the Imports table, and it will recurse on itself for any subordinate modules. The result is code that will likely be difficult for you to follow, but LdrpUpdateLoadCount.cpp contains my attempt to write pseudocode for this procedure. Distilling the essence of the pseudocode leaves the following: LdrpUpdateLoadCount walks through either the Bound Imports Descriptor or the Imports Descriptor looking for imported modules using LdrpCheckForLoadedDll and the NTDLL hash table. If the module was newly loaded by a LoadLibraryExW call, then LdrpUpdateLoadCount updates its reference count and walks its tables for any imports.

You can easily imagine a tree structure that describes the relationships between DLLs and their imports, and LdrpUpdateLoadCount must walk the tree completely to update everyone's reference count correctly. Some modules enter the process with a reference count of -1 and are skipped by this update. I leave here a question for future exploration: why do some DLLs have a reference count of -1 and the others contain an actual count?

Longtime readers of Under the Hood may recall a handy utility Matt Pietrek wrote named NukeDll. Back in the ancient days of 16-bit Windows 3.x, an application that General Protection Faulted had the nasty habit of leaving wreckage strewn about in memory in the form of orphaned DLLs—their reference counts never reached 0 and were not released from the common address space that was part of Windows. Using NukeDll, you could nuke these orphans out of existence and free precious resources. The need for a utility like that has been virtually eliminated with Windows NT and its successors because of the compartmentalization imposed upon processes by the operating system. Still, the reference count is important; your application could be loading and unloading modules but leaking HINSTANCE's because of a mismatch in the LoadLibrary/FreeLibrary pairs. But this will only hurt your own buggy application and you will only chomp through your own resources; other processes in Windows NT and Windows 2000 will be isolated from this behavior.

Once LdrpUpdateLoadCount has finished its work and has returned, LdrpLoadDll checks the return code from LdrpWalkImportDescriptor. If the code is STATUS\_SUCCESS, processing continues on to DLL initialization (which was described in Matt Pietrek's September 1999 Under the Hood column) and is followed by leaving the process's critical section. But if there was a problem in

# What Goes On Inside Windows 2000: Solving the Mysteries of the Loader

Russ Osterlund

LdrpWalkImportDescriptor, then LdrpLoadDll must back out all of the work done up until now, mostly by invoking LdrpUnloadDll. An image base is sent back to LoadLibraryExW in the form of an HMODULE.

## The LOAD\_WITH\_ALTERED\_SEARCH\_PATH Option

There is one scenario that I have not described yet, and that is when a call to LoadLibraryExW is made with dwFlags equal to LOAD\_WITH\_ALTERED\_SEARCH\_PATH. Now that you have grown somewhat accustomed to wandering around DLLs, you might want to experiment on your own with this small wrinkle. Change the Test sample program to issue this version of a LoadLibraryExW call and pay particular attention to the first parameter for LdrpLoadDll and note any differences.

You might also want to create two copies of TestDll, storing one copy in your \TEMP directory, and observing what happens. With a minimum amount of effort you will be able to manufacture a situation where two copies of TestDll are loaded, one from the current working directory and the second from the temp directory. That would demonstrate how the .NET CLR support for side-by-side execution of different versions of the same assembly might be implemented (although this side-by-side capability has been available since at least Windows NT 4.0).

## What You've Learned

Next time the debugger displays a DLL load notification, you will know with some degree of confidence the state that the module is in: it has been mapped into memory, it has not been added to the PEB's housekeeping area, and, most importantly, it has not been initialized yet.

You also have learned that the PEB contains not one, but three lists enumerating loaded modules in load, memory, and initialization order. (There are also many other fields in the PEB worthy of examination since they lead to other vital pieces of information on your process.) As Matt Pietrek has pointed out, the order of the DLLs you see displayed inside the debugger is not the order in which DLLs are initialized, as many people mistakenly believe.

Probably the most important fact to hold onto is that a simple call to LoadLibrary results in many more things occurring under the covers than might initially be apparent. The loader must examine each and every API that DLL imports from other DLLs in order to calculate a real address in memory and perhaps load additional DLLs and check to see if an API may have been forwarded on to another procedure housed in another DLL.

A loading DLL may bring in additional modules where the process just described will be repeated over and over again.

The overhead that all of this processing brings to your application may be reduced by investigating the use of the SDK utility, BIND.EXE. The loader still checks the reference to each DLL contained in your program, but as long as the entries are not stale (in other words, the entries are still correct), the address calculation and forwarded API processing will be safely bypassed.

Finally, you have seen that DLLs are reference-counted, just as they were in the ancient Windows 3.x days. Although this count does not have the same systemwide effect that it once had, a DLL that you are trying to manage dynamically will still produce resource leaks if you have not properly matched up FreeLibrary calls with each LoadLibrary call.

## Conclusion

You should prepare yourself for additional trips into NTDLL.DLL during future debugging sessions because, like LoadLibraryEx, many Kernel32 APIs lead inevitably to undocumented routines that reside in NTDLL. If you end your investigation prematurely because of a reluctance to enter this

# What Goes On Inside Windows 2000: Solving the Mysteries of the Loader

Russ Osterlund

uncharted territory, you may miss the real cause of your bug or, at least, a better understanding of your problem. I plan to maintain the pseudocode at my Web site, <http://www.smidgeonsoft.com>, so if you find any errors or improvements, please pass them along and I will incorporate them into the code listings.

**Figure 1 Loader APIs**

API	NTDLL APIs
LoadResource	LdrAccessResource LdrAlternateResourcesEnabled
DisableThreadLibraryCalls	LdrDisableThreadCalloutsForDll LdrEnumResources LdrFindAppCompatVariableInfo LdrFindEntryForAddress
EnumResourceTypesW	LdrFindResourceDirectory_U
FindResourceExA	LdrFindResource_U LdrFlushAlternateResourceModules LdrGetAlternateResourceModuleHandle
GetModuleHandleForUnicodeString	LdrGetDllHandle
GetProcAddress	LdrGetProcedureAddress LdrInitializeThunk
LoadLibraryEx (LOAD_LIBRARY_AS_DATAFILE)	LdrLoadAlternateResourceModule
LoadLibrary	LdrLoadDll LdrProcessRelocationBlock LdrQueryApplicationCompatibilityGoo LdrQueryImageFileExecutionOptions LdrQueryProcessModuleInformation LdrRelocateImage
ExitProcess	LdrShutdownProcess
ExitThread	LdrShutdownThread LdrUnloadAlternateResourceModule
FreeLibrary	LdrUnloadDll LdrVerifyImageMatchesChecksum LdrVerifyMappedImageMatchesChecksum

**Figure 2 Private Loader APIs**

```

LdrpAccessResourceData
LdrpAllocateDataTableEntry
LdrpAllocateTls
LdrpCallInitRoutine
LdrpCallTlsInitializers
LdrpCheckForKnownDll
LdrpCheckForLoadedDll
LdrpCheckForLoadedDllHandle
LdrpClearLoadInProgress
LdrpCompareResourceNames_U
LdrpCreateDllSection
LdrpDefineDllTag
LdrpDllTagProcedures
LdrpDphDetectSnapRoutines
LdrpDphInitializeTargetDll
LdrpDphSnapImports
LdrpFetchAddressOfEntryPoint
LdrpForkProcess
LdrpFreeTls
LdrpGetProcedureAddress
LdrpInitializationFailure
LdrpInitialize
LdrpInitializeProcess
LdrpInitializeThread
LdrpInitializeTls
    
```

# What Goes On Inside Windows 2000: Solving the Mysteries of the Loader

Russ Osterlund

```
LdrpInsertMemoryTableEntry
LdrpLoadDll
LdrpLoadImportModule
LdrpMapDll
LdrpNameToOrdinal
LdrpRelocateStartContext
LdrpResolveDllName
LdrpRunInitializeRoutines
LdrpSearchResourceSection_U
LdrpSetAlternateResourceModuleHandle
LdrpSetProtection
LdrpSnapIAT
LdrpSnapThunk
LdrpTagAllocateHeap0...LdrpTagAllocateHeap63
LdrpTagAllocateHeap
LdrpUpdateLoadCount
LdrpValidateImageForMp
LdrpWalkImportDescriptor
```

**Figure 3 Internal Loader Routines**

#1 - LoadLibraryExW 3rd parameter is 0:

```
LdrLoadDll (0x77f889a9)
  LdrpLoadDll 0x77f887e0
    LdrpCheckForLoadedDll (0x77f87122)
    LdrpMapDll (0x77f8bc77)
      LdrpCheckForKnownDll (0x77f8c62b)
      LdrpResolveDllName (0x77f8c3df)
      LdrpCreateDllSection (0x77f8c355)
      LdrpAllocateDataTableEntry (0x77f8be69)
      LdrpFetchAddressOfEntryPoint (0x77f8bf23)
      LdrpInsertMemoryTableEntry (0x77f8bebb)
    LdrpWalkImportDescriptor (0x77f8be15)
      LdrpLoadImportModule (0x77f8bfd1)
        *LdrpCheckForLoadedDll
      LdrpSnapIAT (0x77f8c047)
        LdrpSnapThunk (0x77f87bd1)
          LdrpNameToOrdinal (0x77f87cf0)
          **LdrpLoadDll
          LdrpGetProcedureAddress (0x77f87a20)
            LdrpCheckforLoadedDllHandle (0x77f870cc)
            **LdrpSnapThunk
        LdrpUpdateLoadCount (0x77f88afa)
          *LdrpCheckForLoadedDll
          **LdrpUpdateLoadCount
      LdrpRunInitializeRoutines (0x77f8bcb8)
      LdrpClearLoadInProgress (0x77f88c12)
```

#2 - LoadLibraryExW 3rd parameter is 0 and DLL has been bound:

```
LdrLoadDll (0x77f889a9)
  LdrpLoadDll 0x77f887e0
    LdrpCheckForLoadedDll (0x77f87122)
    LdrpMapDll (0x77f8bc77)
      LdrpCheckForKnownDll (0x77f8c62b)
      LdrpResolveDllName (0x77f8c3df)
      LdrpCreateDllSection (0x77f8c355)
      LdrpAllocateDataTableEntry (0x77f8be69)
      LdrpFetchAddressOfEntryPoint (0x77f8bf23)
      LdrpInsertMemoryTableEntry (0x77f8bebb)
    LdrpWalkImportDescriptor (0x77f8be15)
      LdrpLoadImportModule (0x77f8bfd1)
        *LdrpCheckForLoadedDll
      LdrpUpdateLoadCount (0x77f88afa)
```

# What Goes On Inside Windows 2000: Solving the Mysteries of the Loader

Russ Osterlund

```
*LdrpCheckForLoadedDll
**LdrpUpdateLoadCount
LdrpRunInitializeRoutines (0x77f8bcb8)
LdrpClearLoadInProgress (0x77f88c12)

#3 - LoadLibraryExW 3rd Parameter is DONT_RESOLVE_DLL_REFERENCES:
LdrLoadDll (0x77f889a9)
  LdrpLoadDll 0x77f887e0
    LdrpCheckForLoadedDll (0x77f87122)
    LdrpMapDll (0x77f8bc77)
      LdrpCheckForKnownDll (0x77f8c62b)
      LdrpResolveDllName (0x77f8c3df)
      LdrpCreateDllSection (0x77f8c355)
      LdrpAllocateDataTableEntry (0x77f8be69)
      LdrpFetchAddressOfEntryPoint (0x77f8bf23)
      LdrpInsertMemoryTableEntry (0x77f8bebb)

#4 - LoadLibraryExW 3rd Parameter is LOAD_WITH_ALTERED_SEARCH_PATH:
LdrLoadDll (0x77f889a9)
  LdrpLoadDll 0x77f887e0
    LdrpCheckForLoadedDll (0x77f87122)
    LdrpMapDll (0x77f8bc77)
      LdrpCheckForKnownDll (0x77f8c62b)
      LdrpResolveDllName (0x77f8c3df)
      LdrpCreateDllSection (0x77f8c355)
      LdrpAllocateDataTableEntry (0x77f8be69)
      LdrpFetchAddressOfEntryPoint (0x77f8bf23)
      LdrpInsertMemoryTableEntry (0x77f8bebb)
    LdrpWalkImportDescriptor (0x77f8be15)
      LdrpLoadImportModule (0x77f8bfd1)
        *LdrpCheckForLoadedDll
    LdrpUpdateLoadCount (0x77f88afa)
      *LdrpCheckForLoadedDll
      **LdrpUpdateLoadCount
    LdrpRunInitializeRoutines (0x77f8bcb8)
    LdrpClearLoadInProgress (0x77f88c12)

#5 - LoadLibraryExW 3rd Parameter is LOAD_LIBRARY_AS_DATAFILE:
LdrpCheckForLoadedDll (0x77f87122)
```

All addresses based upon Windows 2000 Professional (Build 2195: Service Pack 1)

\* previously documented

\*\* recursive call

**Figure 4 APIs Forwarded to NTDLL**

API	Destination
DeleteCriticalSection	Forwarded to NTDLL.RtlDeleteCriticalSection
EnterCriticalSection	Forwarded to NTDLL.RtlEnterCriticalSection
HeapAlloc	Forwarded to NTDLL.RtlAllocateHeap
HeapFree	Forwarded to NTDLL.RtlFreeHeap
HeapReAlloc	Forwarded to NTDLL.RtlReAllocateHeap
HeapSize	Forwarded to NTDLL.RtlSizeHeap
LeaveCriticalSection	Forwarded to NTDLL.RtlLeaveCriticalSection
RtlFillMemory	Forwarded to NTDLL.RtlFillMemory
RtlMoveMemory	Forwarded to NTDLL.RtlMoveMemory
RtlUnwind	Forwarded to NTDLL.RtlUnwind
RtlZeroMemory	Forwarded to NTDLL.RtlZeroMemory
SetCriticalSectionSpinCount	Forwarded to NTDLL.RtlSetCriticalSection- SpinCount

# What Goes On Inside Windows 2000: Solving the Mysteries of the Loader

Russ Osterlund

TryEnterCriticalSection	Forwarded to NTDLL.RtlTryEnterCriticalSection
VerSetConditionMask	Forwarded to NTDLL.VerSetConditionMask

**Figure 5** Binding via LdrpSnapIAT and LdrpSnapThunk for Forwarded.DLL

Address	IAT File	IAT Memory	API Name (* - Forwarded APIs)
1200B000	0000C314 =>	77E851E1	HeapCreate
* 1200B004	0000C322 =>	77FCA535	HeapFree (RtlFreeHeap)
1200B008	0000C0AA =>	77E8F07F	GetCommandLineA
1200B00C	0000C0BC =>	77E85C77	GetVersion
1200B010	0000C0CA =>	77EA83DE	DbgBreakPoint
1200B014	0000C0D8 =>	77E8F043	GetStdHandle
1200B018	0000C0E8 =>	77E8334F	WriteFile
1200B01C	0000C0F4 =>	77E82EF1	InterlockedDecrement
1200B020	0000C10C =>	77E9E0C8	OutputDebugStringA
1200B024	0000C122 =>	77E87031	GetProcAddress
1200B028	0000C134 =>	77E87273	LoadLibraryA
1200B02C	0000C144 =>	77E82EE0	InterlockedIncrement
1200B030	0000C15C =>	77E88885	GetModuleFileNameA
1200B034	0000C172 =>	77E8F32D	ExitProcess
1200B038	0000C180 =>	77EB45FF	TerminateProcess
1200B03C	0000C194 =>	77E8304F	GetCurrentProcess
1200B040	0000C1A8 =>	77E83510	GetCurrentThreadId
1200B044	0000C1BE =>	77E836DD	TlsSetValue
1200B048	0000C1CC =>	77E8C512	TlsAlloc
1200B04C	0000C1D8 =>	77E8F254	TlsFree
1200B050	0000C1E2 =>	77E83008	SetLastError
1200B054	0000C1F2 =>	77E83025	TlsGetValue
1200B058	0000C200 =>	77E8301B	GetLastError
1200B05C	0000C210 =>	77E831E7	SetHandleCount
1200B060	0000C222 =>	77E84C93	GetFileType
1200B064	0000C230 =>	77E8F10A	GetStartupInfoA
* 1200B068	0000C242 =>	77F837C4	DeleteCriticalSection (RtlDeleteCriticalSection)
1200B06C	0000C25A =>	77E83A61	IsBadWritePtr
1200B070	0000C26A =>	77E84F4F	IsBadReadPtr
1200B074	0000C27A =>	77E8BC6C	HeapValidate
1200B078	0000C28A =>	77E82116	FreeEnvironmentStringsA
1200B07C	0000C2A4 =>	77E8F085	FreeEnvironmentStringsW
1200B080	0000C2BE =>	77E8593F	WideCharToMultiByte
1200B084	0000C2D4 =>	77E9C60D	GetEnvironmentStrings
1200B088	0000C2EC =>	77E8324E	GetEnvironmentStringsW
1200B08C	0000C306 =>	77E8523C	HeapDestroy
1200B090	0000C41E =>	77E841B6	LCMapStringW
1200B094	0000C40E =>	77E95278	LCMapStringA
1200B098	0000C32E =>	77E85194	VirtualFree
1200B09C	0000C33C =>	77E83833	InitializeCriticalSection
* 1200B0A0	0000C358 =>	77F81B42	EnterCriticalSection (RtlEnterCriticalSection)
* 1200B0A4	0000C370 =>	77F81B73	LeaveCriticalSection (RtlLeaveCriticalSection)
* 1200B0A8	0000C388 =>	77FCA055	HeapAlloc (RtlAllocateHeap)
* 1200B0AC	0000C394 =>	77F85B48	HeapReAlloc (RtlReAllocateHeap)
1200B0B0	0000C3A2 =>	77E850EC	VirtualAlloc
1200B0B4	0000C3B2 =>	77E8EFB8	GetCPInfo
1200B0B8	0000C3BE =>	77E83852	GetACP
1200B0BC	0000C3C8 =>	77E8724D	GetOEMCP
1200B0C0	0000C3D4 =>	77E84035	MultiByteToWideChar
1200B0C4	0000C3EA =>	77E9740C	GetStringTypeA
1200B0C8	0000C3FC =>	77E867D4	GetStringTypeW
1200B0CC	0000C42E =>	77E853E8	SetFilePointer
* 1200B0D0	0000C440 =>	77F8E13A	RtlUnwind (RtlUnwind)
1200B0D4	0000C44C =>	77E8F3BC	SetStdHandle
1200B0D8	0000C45C =>	77E863C1	FlushFileBuffers

# What Goes On Inside Windows 2000: Solving the Mysteries of the Loader

Russ Osterlund

```
1200B0DC 0000C470 => 77E83053 CloseHandle
1200B0E0 00000000      00000000
1200B0E4 0000C090 => 77E1F098 MessageBoxW
1200B0E8 00000000      00000000
```

\*\*\*\*\*  
LdrSnap Display (Note the displays of forwarded APIs):

```
LDR: LdrLoadDll, loading Forwarder.DLL from
      E:\PROJECTS\TEMP\Test\debug;. ;D:\WINNT\System32;D:\WINNT\system;D:\WINNT;...
LDR: Loading (DYNAMIC) E:\PROJECTS\TEMP\Test\debug\Forwarder.DLL
LDR: KERNEL32.dll used by Forwarder.DLL
LDR: Snapping imports for Forwarder.DLL from KERNEL32.dll
LDR: LdrLoadDll, loading NTDLL.dll from
LDR: LdrGetProcedureAddress by NAME - RtlEnterCriticalSection
LDR: LdrLoadDll, loading NTDLL.dll from
LDR: LdrGetProcedureAddress by NAME - RtlDeleteCriticalSection
LDR: LdrLoadDll, loading NTDLL.dll from
LDR: LdrGetProcedureAddress by NAME - RtlFreeHeap
LDR: LdrLoadDll, loading NTDLL.dll from
LDR: LdrGetProcedureAddress by NAME - RtlLeaveCriticalSection
LDR: LdrLoadDll, loading NTDLL.dll from
LDR: LdrGetProcedureAddress by NAME - RtlAllocateHeap
LDR: LdrLoadDll, loading NTDLL.dll from
LDR: LdrGetProcedureAddress by NAME - RtlReAllocateHeap
LDR: LdrLoadDll, loading NTDLL.dll from
LDR: LdrGetProcedureAddress by NAME - RtlUnwind
LDR: Real INIT LIST
      E:\PROJECTS\TEMP\Test\debug\Forwarder.DLL init routine 110010e9
LDR: Forwarder.DLL loaded. - Calling init routine at 110010e9
```

**Figure 6 Pre-binding using SDK Bind**

Address	IAT File & Memory	API Name (* - Forwarded APIs)
10004000	77E851E1	HeapCreate
10004004	77E85C77	GetVersion
10004008	77E8F32D	ExitProcess
1000400C	77EB45FF	TerminateProcess
10004010	77E8304F	GetCurrentProcess
10004014	77E83510	GetCurrentThreadId
10004018	77E836DD	TlsSetValue
1000401C	77E8C512	TlsAlloc
10004020	77E8F254	TlsFree
10004024	77E83025	TlsGetValue
10004028	77E831E7	SetHandleCount
1000402C	77E8F043	GetStdHandle
10004030	77E84C93	GetFileType
10004034	77E8F10A	GetStartupInfoA
* 10004038	77F837C4	DeleteCriticalSection (RtlDeleteCriticalSection)
1000403C	77E88885	GetModuleFileNameA
10004040	77E82116	FreeEnvironmentStringsA
10004044	77E8F085	FreeEnvironmentStringsW
10004048	77E8593F	WideCharToMultibyte
1000404C	77E9C60D	GetEnvironmentStrings
10004050	77E8324E	GetEnvironmentStringsW
10004054	77E8523C	HeapDestroy
10004058	77E8F07F	GetCommandLineA
1000405C	77E85194	VirtualFree
* 10004060	77FCA535	HeapFree (RtlFreeHeap)
10004064	77E8334F	WriteFile
10004068	77E83833	InitializeCriticalSection

# What Goes On Inside Windows 2000: Solving the Mysteries of the Loader

Russ Osterlund

```
* 1000406C 77F81B42 EnterCriticalSection (RtlEnterCriticalSection)
* 10004070 77F81B73 LeaveCriticalSection (RtlLeaveCriticalSection)
* 10004074 77FCA055 HeapAlloc (RtlAllocateHeap)
  10004078 77E8EFB8 GetCPInfo
  1000407C 77E83852 GetACP
  10004080 77E8724D GetOEMCP
  10004084 77E850EC VirtualAlloc
* 10004088 77F85B48 HeapReAlloc (RtlReAllocateHeap)
  1000408C 77E87031 GetProcAddress
  10004090 77E87273 LoadLibraryA
  10004094 77E84035 MultiByteToWideChar
  10004098 77E95278 LCMapStringA
  1000409C 77E841B6 LCMapStringW
  100040A0 77E9740C GetStringTypeA
  100040A4 77E867D4 GetStringTypeW
* 100040A8 77F8E13A RtlUnwind (RtlUnwind)
```

```
*****
LdrSnap Display (Note there are no displays of forwarded APIs):
```

```
LDR: LdrLoadDll, loading TestDll.DLL from
      E:\PROJECTS\TEMP\Test\debug;. ;D:\WINNT\System32;D:\WINNT\system;D:\WINNT;...
LDR: Loading (DYNAMIC) E:\PROJECTS\TEMP\Test\debug\TestDll.DLL
LDR: TestDll.DLL bound to KERNEL32.dll
LDR: TestDll.DLL has correct binding to KERNEL32.dll
LDR: TestDll.DLL bound to NTDLL.DLL via forwarder(s) from KERNEL32.dll
LDR: TestDll.DLL has correct binding to NTDLL.DLL
LDR: Real INIT LIST
      E:\PROJECTS\TEMP\Test\debug\TestDll.DLL init routine 10001109
LDR: TestDll.DLL loaded. - Calling init routine at 10001109
```